Common Lisp Object System Specification

1. Programmer Interface Concepts

Authors: Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel,
Sonya E. Keene, Gregor Kiczales, and David A. Moon.

Draft Dated: June 15, 1988

For information about obtaining the sources for this document, send an Internet message to common-lisp-object-system-specification-request@sail.stanford.edu.

The authors wish to thank Patrick Dussud, Kenneth Kahn, Jim Kempf, Larry Masinter, Mark Stefik, Daniel L. Weinreb, and Jon L White for their contributions to this document.

At the X3J13 meeting on June 15, 1988, the following motion was adopted:

"The X3J13 Committee hereby accepts chapters 1 and 2 of the Common Lisp Object System, as defined in document 88-002R, for inclusion in the Common Lisp language being specified by this committee. Subsequent changes will be handled through the usual editorial and cleanup processes."

CONTENTS

Introduction

The Common Lisp Object System is an object-oriented extension to Common Lisp as defined in *Common Lisp: The Language*, by Guy L. Steele Jr. It is based on generic functions, multiple inheritance, declarative method combination, and a meta-object protocol.

The first two chapters of this specification present a description of the standard Programmer Interface for the Common Lisp Object System. The first chapter contains a description of the concepts of the Common Lisp Object System, and the second contains a description of the functions and macros in the Common Lisp Object System Programmer Interface. The chapter "The Common Lisp Object System Meta-Object Protocol" describes how the Common Lisp Object System can be customized.

The fundamental objects of the Common Lisp Object System are classes, instances, generic functions, and methods.

A ***class*** object determines the structure and behavior of a set of other objects, which are called its ***instances***. Every Common Lisp object is an ***instance*** of a class. The class of an object determines the set of operations that can be performed on the object.

A ***generic function*** is a function whose behavior depends on the classes or identities of the arguments supplied to it. A generic function object contains a set of methods, a lambda-list, a method combination type, and other information. The ***methods*** define the class-specific behavior and operations of the generic function; a method is said to ***specialize*** a generic function. When invoked, a generic function executes a subset of its methods based on the classes of its arguments.

A generic function can be used in the same ways that an ordinary function can be used in Common Lisp; in particular, a generic function can be used as an argument to **funcall** and **apply** and can be given a global or a local name.

A ***method*** is an object that contains a method function, a sequence of ***parameter specializers*** that specify when the given method is applicable, and a sequence of ***qualifiers*** that is used by the ***method combination*** facility to distinguish among methods. Each required formal parameter of each method has an associated parameter specializer, and the method will be invoked only on arguments that satisfy its parameter specializers.

The method combination facility controls the selection of methods, the order in which they are run, and the values that are returned by the generic function. The Common Lisp Object System offers a default method combination type and provides a facility for declaring new types of method combination.

Error Terminology

The terminology used in this document to describe erroneous situations differs from the terminology used in *Common Lisp: The Language*, by Guy L. Steele Jr. This terminology involves ***situations***; a situation is the evaluation of an expression in some specific context. For example, a situation might be the invocation of a function on arguments that fail to satisfy some specified constraints.

In the specification of the Common Lisp Object System, the behavior of programs in all situations is described, and the options available to the implementor are defined. No implementation is allowed to extend the syntax or semantics of the Object System except as explicitly defined in the Object System specification. In particular, no implementation is allowed to extend the syntax of the Object System in such a way that ambiguity between the specified syntax of Object System and those extensions is possible.

"When situation S occurs, an error is signaled."

This terminology has the following meaning:

- If this situation occurs, an error will be signaled in the interpreter and in code compiled under all compiler safety optimization levels.
- Valid programs may rely on the fact that an error will be signaled in the interpreter and in code compiled under all compiler safety optimization levels.
- Every implementation is required to detect such an error in the interpreter and in code compiled under all compiler safety optimization levels.

"When situation S occurs, an error should be signaled."

This terminology has the following meaning:

- If this situation occurs, an error will be signaled at least in the interpreter and in code compiled under the safest compiler safety optimization level.
- Valid programs may not rely on the fact that an error will be signaled.
- Every implementation is required to detect such an error at least in the interpreter and in code compiled under the safest compiler safety optimization level.
- When an error is not signaled, the results are undefined (see below).

"When situation S occurs, the results are undefined."

This terminology has the following meaning:

- If this situation occurs, the results are unpredictable. The results may range from harmless to fatal.

- Implementations are allowed to detect this situation and signal an error, but no implementation is required to detect the situation.
- No valid program may depend on the effects of this situation, and all valid programs are required to treat the effects of this situation as unpredictable.

"When situation *S* occurs, the results are unspecified."

This terminology has the following meaning:

- The effects of this situation are not specified in the Object System, but the effects are harmless.
- Implementations are allowed to specify the effects of this situation.
- No portable program can depend on the effects of this situation, and all portable programs are required to treat the situation as unpredictable but harmless.

"The Common Lisp Object System may be extended to cover situation *S*."

The meaning of this terminology is that an implementation is free to treat situation *S* in one of three ways:

- When situation *S* occurs, an error is signaled at least in the interpreter and in code compiled under the safest compiler safety optimization level.
- When situation *S* occurs, the results are undefined.
- When situation *S* occurs, the results are defined and specified.

In addition, this terminology has the following meaning:

- No portable program can depend on the effects of this situation, and all portable programs are required to treat the situation as undefined.

"Implementations are free to extend the syntax *S*."

This terminology has the following meaning:

- Implementations are allowed to define unambiguous extensions to syntax *S*.
- No portable program can depend on this extension, all portable programs are required to treat the syntax as meaningless.

The Common Lisp Object System specification may disallow certain extensions while allowing others.

Classes

A ***class*** is an object that determines the structure and behavior of a set of other objects, which are called its ***instances***.

A class can inherit structure and behavior from other classes. A class whose definition refers to other classes for the purpose of inheriting from them is said to be a ***subclass*** of each of those classes. The classes that are designated for purposes of inheritance are said to be ***superclasses*** of the inheriting class.

A class can have a ***name***. The function **class-name** takes a class object and returns its name. The name of an anonymous class is **nil**. A symbol can ***name*** a class. The function **find-class** takes a symbol and returns the class that the symbol names. A class has a ***proper name*** if the name is a symbol and if the name of the class names that class. That is, a class C has the ***proper name*** S if $S =$ `(class-name` C`)` and $C =$ `(find-class` S`)`. Notice that it is possible for `(find-class` S_1`)` = `(find-class` S_2`)` and $S_1 \neq S_2$. If $C =$ `(find-class` S`)`, we say that C is the ***class named*** S.

A class C_1 is a ***direct superclass*** of a class C_2 if C_2 explicitly designates C_1 as a superclass in its definition. In this case C_2 is a ***direct subclass*** of C_1. A class C_n is a ***superclass*** of a class C_1 if there exists a series of classes $C_2, \ldots, C_{n-1}$ such that C_{i+1} is a direct superclass of C_i for $1 \leq i < n$. In this case, C_1 is a ***subclass*** of C_n. A class is considered neither a superclass nor a subclass of itself. That is, if C_1 is a superclass of C_2, then $C_1 \neq C_2$. The set of classes consisting of some given class C along with all of its superclasses is called "C and its superclasses."

Each class has a ***class precedence list***, which is a total ordering on the set of the given class and its superclasses. The total ordering is expressed as a list ordered from most specific to least specific. The class precedence list is used in several ways. In general, more specific classes can ***shadow***, or override, features that would otherwise be inherited from less specific classes. The method selection and combination process uses the class precedence list to order methods from most specific to least specific.

When a class is defined, the order in which its direct superclasses are mentioned in the defining form is important. Each class has a ***local precedence order***, which is a list consisting of the class followed by its direct superclasses in the order mentioned in the defining form.

A class precedence list is always consistent with the local precedence order of each class in the list. The classes in each local precedence order appear within the class precedence list in the same order. If the local precedence orders are inconsistent with each other, no class precedence list can be constructed, and an error is signaled. The class precedence list and its computation is discussed in the section "Determining the Class Precedence List."

Classes are organized into a ***directed acyclic graph***. There are two distinguished classes, named **t** and **standard-object**. The class named **t** has no superclasses. It is a superclass of every class except itself. The class named **standard-object** is an instance of the class **standard-class** and is a superclass of every class that is an instance of **standard-class** except itself.

There is a mapping from the Common Lisp Object System class space into the Common Lisp type space. Many of the standard Common Lisp types specified in *Common Lisp: The Language* have a corresponding class that has the same name as the type. Some Common Lisp types do not have a corresponding class. The integration of the type and class systems is discussed in the section "Integrating Types and Classes."

Classes are represented by objects that are themselves instances of classes. The class of the class of an object is termed the ***metaclass*** of that object. When no misinterpretation is possible, the term ***metaclass*** will be used to refer to a class that has instances that are themselves classes. The metaclass determines the form of inheritance used by the classes that are its instances and the representation of the instances of those classes. The Common Lisp Object System provides a default metaclass, **standard-class**, that is appropriate for most programs. The meta-object protocol provides mechanisms for defining and using new metaclasses.

Except where otherwise specified, all classes mentioned in this chapter are instances of the class **standard-class**, all generic functions are instances of the class **standard-generic-function**, and all methods are instances of the class **standard-method**.

Defining Classes

The macro **defclass** is used to define a new named class. The syntax for **defclass** is given in Figure 2-1.

The definition of a class includes:

- The name of the new class. For newly defined classes this name is a proper name.
- The list of the direct superclasses of the new class.
- A set of ***slot specifiers***. Each slot specifier includes the name of the slot and zero or more ***slot options***. A slot option pertains only to a single slot. If a class definition contains two slot specifiers with the same name, an error is signaled.
- A set of ***class options***. Each class option pertains to the class as a whole.

The slot options and class options of the **defclass** form provide mechanisms for the following:

- Supplying a default initial value form for a given slot.
- Requesting that methods for generic functions be automatically generated for reading or writing slots.
- Controlling whether a given slot is shared by instances of the class or whether each instance of the class has its own slot.
- Supplying a set of initialization arguments and initialization argument defaults to be used in instance creation.

- Indicating that the metaclass is to be other than the default.
- Indicating the expected type for the value stored in the slot.
- Indicating the documentation string for the slot.

Creating Instances of Classes

The generic function **make-instance** creates and returns a new instance of a class. The Object System provides several mechanisms for specifying how a new instance is to be initialized. For example, it is possible to specify the initial values for slots in newly created instances either by giving arguments to **make-instance** or by providing default initial values. Further initialization activities can be performed by methods written for generic functions that are part of the initialization protocol. The complete initialization protocol is described in the section "Object Creation and Initialization."

Slots

An object that has **standard-class** as its metaclass has zero or more named slots. The slots of an object are determined by the class of the object. Each slot can hold one value. The name of a slot is a symbol that is syntactically valid for use as a Common Lisp variable name.

When a slot does not have a value, the slot is said to be ***unbound***. When an unbound slot is read, the generic function **slot-unbound** is invoked. The system-supplied primary method for **slot-unbound** signals an error.

The default initial value form for a slot is defined by the **:initform** slot option. When the **:initform** form is used to supply a value, it is evaluated in the lexical environment in which the **defclass** form was evaluated. The **:initform** along with the lexical environment in which the **defclass** form was evaluated is called a ***captured* :initform**. See the section "Object Creation and Initialization" for more details.

A ***local slot*** is defined to be a slot that is visible to exactly one instance, namely the one in which the slot is allocated. A ***shared slot*** is defined to be a slot that is visible to more than one instance of a given class and its subclasses.

A class is said to ***define*** a slot with a given name when the **defclass** form for that class contains a slot specifier with that name. Defining a local slot does not immediately create a slot; it causes a slot to be created each time an instance of the class is created. Defining a shared slot immediately creates a slot.

The **:allocation** slot option to **defclass** controls the kind of slot that is defined. If the value of the **:allocation** slot option is **:instance**, a local slot is created. If the value of **:allocation** is **:class**, a shared slot is created.

A slot is said to be ***accessible*** in an instance of a class if the slot is defined by the class of the instance or is inherited from a superclass of that class. At most one slot of a given name can be accessible in an instance. A shared slot defined by a class is accessible in all instances of that class. A detailed explanation of the inheritance of slots is given in the section "Inheritance of Slots and Slot Options."

Accessing Slots

Slots can be accessed in two ways: by use of the primitive function **slot-value** and by use of generic functions generated by the **defclass** form.

The function **slot-value** can be used with any of the slot names specified in the **defclass** form to access a specific slot accessible in an instance of the given class.

The macro **defclass** provides syntax for generating methods to read and write slots. If a ***reader*** is requested, a method is automatically generated for reading the value of the slot, but no method for storing a value into it is generated. If a ***writer*** is requested, a method is automatically generated for storing a value into the slot, but no method for reading its value is generated. If an ***accessor*** is requested, a method for reading the value of the slot and a method for storing a value into the slot are automatically generated. Reader and writer methods are implemented using **slot-value**.

When a reader or writer is specified for a slot, the name of the generic function to which the generated method belongs is directly specified. If the name specified for the writer option is the symbol *name*, the name of the generic function for writing the slot is the symbol *name*, and the generic function takes two arguments: the new value and the instance, in that order. If the name specified for the accessor option is the symbol *name*, the name of the generic function for reading the slot is the symbol *name*, and the name of the generic function for writing the slot is the list `(setf` *name*`)`.

A generic function created or modified by supplying reader, writer, or accessor slot options can be treated exactly as an ordinary generic function.

Note that **slot-value** can be used to read or write the value of a slot whether or not reader or writer methods exist for that slot. When **slot-value** is used, no reader or writer methods are invoked.

The macro **with-slots** can be used to establish a lexical environment in which specified slots are lexically available as if they were variables. The macro **with-slots** invokes the function **slot-value** to access the specified slots.

The macro **with-accessors** can be used to establish a lexical environment in which specified slots are lexically available through their accessors as if they were variables. The macro **with-accessors** invokes the appropriate accessors to access the specified slots. Any accessors specified by **with-accessors** must already have been defined before they are used.

Inheritance

A class can inherit methods, slots, and some **defclass** options from its superclasses. The following sections describe the inheritance of methods, the inheritance of slots and slot options, and the inheritance of class options.

Inheritance of Methods

A subclass inherits methods in the sense that any method applicable to all instances of a class is also applicable to all instances of any subclass of that class.

The inheritance of methods acts the same way regardless of whether the method was created by using one of the method-defining forms or by using one of the **defclass** options that causes methods to be generated automatically.

The inheritance of methods is described in detail in the section "Method Selection and Combination."

Inheritance of Slots and Slot Options

The set of the names of all slots accessible in an instance of a class C is the union of the sets of names of slots defined by C and its superclasses. The ***structure*** of an instance is the set of names of local slots in that instance.

In the simplest case, only one class among C and its superclasses defines a slot with a given slot name. If a slot is defined by a superclass of C, the slot is said to be ***inherited***. The characteristics of the slot are determined by the slot specifier of the defining class. Consider the defining class for a slot S. If the value of the **:allocation** slot option is **:instance**, then S is a local slot and each instance of C has its own slot named S that stores its own value. If the value of the **:allocation** slot option is **:class**, then S is a shared slot, the class that defined S stores the value, and all instances of C can access that single slot. If the **:allocation** slot option is omitted, **:instance** is used.

In general, more than one class among C and its superclasses can define a slot with a given name. In such cases, only one slot with the given name is accessible in an instance of C, and the characteristics of that slot are a combination of the several slot specifiers, computed as follows:

- All the slot specifiers for a given slot name are ordered from most specific to least specific, according to the order in C's class precedence list of the classes that define them. All references to the specificity of slot specifiers immediately below refers to this ordering.
- The allocation of a slot is controlled by the most specific slot specifier. If the most specific slot specifier does not contain an **:allocation** slot option, **:instance** is used. Less specific slot specifiers do not affect the allocation.

- The default initial value form for a slot is the value of the **:initform** slot option in the most specific slot specifier that contains one. If no slot specifier contains an **:initform** slot option, the slot has no default initial value form.
- The contents of a slot will always be of type `(and` $T_1 \ldots T_n$`)` where $T_1 \ldots T_n$ are the values of the **:type** slot options contained in all of the slot specifiers. If no slot specifier contains the **:type** slot option, the contents of the slot will always be of type **t**. The result of attempting to store in a slot a value that does not satisfy the type of the slot is undefined.
- The set of initialization arguments that initialize a given slot is the union of the initialization arguments declared in the **:initarg** slot options in all the slot specifiers.
- The documentation string for a slot is the value of the **:documentation** slot option in the most specific slot specifier that contains one. If no slot specifier contains a **:documentation** slot option, the slot has no documentation string.

A consequence of the allocation rule is that a shared slot can be shadowed. For example, if a class C_1 defines a slot named S whose value for the **:allocation** slot option is **:class**, that slot is accessible in instances of C_1 and all of its subclasses. However, if C_2 is a subclass of C_1 and also defines a slot named S, C_1's slot is not shared by instances of C_2 and its subclasses. When a class C_1 defines a shared slot, any subclass C_2 of C_1 will share this single slot unless the **defclass** form for C_2 specifies a slot of the same name or there is a superclass of C_2 that precedes C_1 in the class precedence list of C_2 that defines a slot of the same name.

A consequence of the type rule is that the value of a slot satisfies the type constraint of each slot specifier that contributes to that slot. Because the result of attempting to store in a slot a value that does not satisfy the type constraint for the slot is undefined, the value in a slot might fail to satisfy its type constraint.

The **:reader**, **:writer**, and **:accessor** slot options create methods rather than define the characteristics of a slot. Reader and writer methods are inherited in the sense described in the section "Inheritance of Methods."

Methods that access slots use only the name of the slot and the type of the slot's value. Suppose a superclass provides a method that expects to access a shared slot of a given name, and a subclass defines a local slot with the same name. If the method provided by the superclass is used on an instance of the subclass, the method accesses the local slot.

Inheritance of Class Options

The **:default-initargs** class option is inherited. The set of defaulted initialization arguments for a class is the union of the sets of initialization arguments specified in the **:default-initargs** class options of the class and its superclasses. When more than one default initial value form is supplied for a given initialization argument, the default initial value form that is used is the one supplied by the class that is most specific according to the class precedence list.

If a given **:default-initargs** class option specifies an initialization argument of the same name more than once, an error is signaled.

Examples

```
(defclass C1 ()
    ((S1 :initform 5.4 :type number)
     (S2 :allocation :class)))

(defclass C2 (C1)
    ((S1 :initform 5 :type integer)
     (S2 :allocation :instance)
     (S3 :accessor C2-S3)))
```

Instances of the class `C1` have a local slot named `S1`, whose default initial value is 5.4 and whose value should always be a number. The class `C1` also has a shared slot named `S2`.

There is a local slot named `S1` in instances of `C2`. The default initial value of `S1` is 5. The value of `S1` will be of type `(and integer number)`. There are also local slots named `S2` and `S3` in instances of `C2`. The class `C2` has a method for `C2-S3` for reading the value of slot `S3`; there is also a method for `(setf C2-S3)` that writes the value of `S3`.

Integrating Types and Classes

The Common Lisp Object System maps the space of classes into the Common Lisp type space. Every class that has a proper name has a corresponding type with the same name.

The proper name of every class is a valid type specifier. In addition, every class object is a valid type specifier. Thus the expression `(typep` *object class*`)` evaluates to true if the class of *object* is *class* itself or a subclass of *class*. The evaluation of the expression `(subtypep` *class1 class2*`)` returns the values **t t** if *class1* is a subclass of *class2* or if they are the same class; otherwise it returns the values **nil t**. If *I* is an instance of some class *C* named *S* and *C* is an instance of **standard-class**, the evaluation of the expression `(type-of` *I*`)` will return *S* if *S* is the proper name of *C*; if *S* is not the proper name of *C*, the expression `(type-of` *I*`)` will return *C*.

Because the names of classes and class objects are type specifiers, they may be used in the special form **the** and in type declarations.

Many but not all of the predefined Common Lisp type specifiers have a corresponding class with the same proper name as the type. These type specifiers are listed in Figure 1-1. For example, the type **array** has a corresponding class named **array**. No type specifier that is a list, such as `(vector double-float 100)`, has a corresponding class. The form **deftype** does not create any classes.

Each class that corresponds to a predefined Common Lisp type specifier can be implemented in one of three ways, at the discretion of each implementation. It can be a ***standard class*** (of the kind defined by **defclass**), a ***structure class*** (defined by **defstruct**), or a ***built-in class*** (implemented in a special, non-extensible way).

A built-in class is one whose instances have restricted capabilities or special representations. Attempting to use **defclass** to define subclasses of a built-in class signals an error. Calling **make-instance** to create an instance of a built-in class signals an error. Calling **slot-value** on an instance of a built-in class signals an error. Redefining a built-in class or using **change-class** to change the class of an instance to or from a built-in class signals an error. However, built-in classes can be used as parameter specializers in methods.

It is possible to determine whether a class is a built-in class by checking the metaclass. A standard class is an instance of **standard-class**, a built-in class is an instance of **built-in-class**, and a structure class is an instance of **structure-class**.

Each structure type created by **defstruct** without using the **:type** option has a corresponding class. This class is an instance of **structure-class**. The **:include** option of **defstruct** creates a direct subclass of the class that corresponds to the included structure.

The purpose of specifying that many of the standard Common Lisp type specifiers have a corresponding class is to enable users to write methods that discriminate on these types. Method selection requires that a class precedence list can be determined for each class.

The hierarchical relationships among the Common Lisp type specifiers are mirrored by relationships among the classes corresponding to those types. The existing type hierarchy is used for determining the class precedence list for each class that corresponds to a predefined Common Lisp type. In some cases, *Common Lisp: The Language* does not specify a local precedence order for two supertypes of a given type specifier. For example, **null** is a subtype of both **symbol** and **list**, but *Common Lisp: The Language* does not specify whether **symbol** is more specific or less specific than **list**. The Common Lisp Object System specification defines those relationships for all such classes.

The following figure lists the set of classes required by the Object System that correspond to predefined Common Lisp type specifiers. The superclasses of each such class are presented in order from most specific to most general, thereby defining the class precedence list for the class. The local precedence order for each class that corresponds to a Common Lisp type specifier can be derived from this table.

Predefined Common Lisp Type	**Class Precedence List for Corresponding Class**
array	(array t)
bit-vector	(bit-vector vector array sequence t)
character	(character t)
complex	(complex number t)
cons	(cons list sequence t)
float	(float number t)
integer	(integer rational number t)
list	(list sequence t)
null	(null symbol list sequence t)
number	(number t)
ratio	(ratio rational number t)
rational	(rational number t)
sequence	(sequence t)
string	(string vector array sequence t)
symbol	(symbol t)
t	(t)
vector	(vector array sequence t)

Figure 1–1.

Individual implementations may be extended to define other type specifiers to have a corresponding class. Individual implementations can be extended to add other subclass relationships and to add other elements to the class precedence lists in the above table, as long as they do not violate the type relationships and disjointness requirements specified by *Common Lisp: The Language*. A standard class defined with no direct superclasses is guaranteed to be disjoint from all of the classes in the table, except for the class named **t**.

The following Common Lisp types will have corresponding classes when Common Lisp is modified to define them each as being disjoint from **cons**, **symbol**, **array**, **number**, and **character**:

- function
- hash-table
- package
- pathname
- random-state
- readtable
- stream

Determining the Class Precedence List

The **defclass** form for a class provides a total ordering on that class and its direct superclasses. This ordering is called the ***local precedence order***. It is an ordered list of the class and its direct superclasses. The ***class precedence list*** for a class C is a total ordering on C and its superclasses that is consistent with the local precedence orders for each of C and its superclasses.

A class precedes its direct superclasses, and a direct superclass precedes all other direct superclasses specified to its right in the superclasses list of the **defclass** form. For every class C, define

$$R_C = \{(C, C_1), (C_1, C_2), \ldots, (C_{n-1}, C_n)\}$$

where $C_1, \ldots, C_n$ are the direct superclasses of C in the order in which they are mentioned in the **defclass** form. These ordered pairs generate the total ordering on the class C and its direct superclasses.

Let S_C be the set of C and its superclasses. Let R be

$$R = \bigcup_{c \in S_C} R_c$$

The set R may or may not generate a partial ordering, depending on whether the R_c, $c \in S_C$, are consistent; it is assumed that they are consistent and that R generates a partial ordering. When the R_c are not consistent, it is said that R is inconsistent.

To compute the class precedence list for C, topologically sort the elements of S_C with respect to the partial ordering generated by R. When the topological sort must select a class from a set of two or more classes, none of which are preceded by other classes with respect to R, the class selected is chosen deterministically, as described below.

If R is inconsistent, an error is signaled.

Topological Sorting

Topological sorting proceeds by finding a class C in S_C such that no other class precedes that element according to the elements in R. The class C is placed first in the result. Remove C from S_C, and remove all pairs of the form (C, D), $D \in S_C$, from R. Repeat the process, adding classes with no predecessors to the end of the result. Stop when no element can be found that has no predecessor.

If S_C is not empty and the process has stopped, the set R is inconsistent. If every class in the finite set of classes is preceded by another, then R contains a loop. That is, there is a chain of classes $C_1, \ldots, C_n$ such that C_i precedes C_{i+1}, $1 \leq i < n$, and C_n precedes C_1.

Sometimes there are several classes from S_C with no predecessors. In this case select the one that has a direct subclass rightmost in the class precedence list computed so far. Because a direct

superclass precedes all other direct superclasses to its right, there can be only one such candidate class. If there is no such candidate class, R does not generate a partial ordering—the R_c, $c \in S_C$, are inconsistent.

In more precise terms, let $\{N_1, \ldots, N_m\}$, $m \geq 2$, be the classes from S_C with no predecessors. Let $(C_1 \ldots C_n)$, $n \geq 1$, be the class precedence list constructed so far. C_1 is the most specific class, and C_n is the least specific. Let $1 \leq j \leq n$ be the largest number such that there exists an i where $1 \leq i \leq m$ and N_i is a direct superclass of C_j; N_i is placed next.

The effect of this rule for selecting from a set of classes with no predecessors is that the classes in a simple superclass chain are adjacent in the class precedence list and that classes in each relatively separated subgraph are adjacent in the class precedence list. For example, let T_1 and T_2 be subgraphs whose only element in common is the class J. Suppose that no superclass of J appears in either T_1 or T_2. Let C_1 be the bottom of T_1; and let C_2 be the bottom of T_2. Suppose C is a class whose direct superclasses are C_1 and C_2 in that order, then the class precedence list for C will start with C and will be followed by all classes in T_1 except J. All the classes of T_2 will be next. The class J and its superclasses will appear last.

Examples

This example determines a class precedence list for the class `pie`. The following classes are defined:

```
(defclass pie (apple cinnamon) ())

(defclass apple (fruit) ())

(defclass cinnamon (spice) ())

(defclass fruit (food) ())

(defclass spice (food) ())

(defclass food () ())
```

The set $S = \{$`pie, apple, cinnamon, fruit, spice, food, standard-object, t`$\}$. The set $R =$ $\{$`(pie, apple), (apple, cinnamon), (apple, fruit), (cinnamon, spice), (fruit, food), (spice, food), (food, standard-object), (standard-object, t)`$\}$.

The class `pie` is not preceded by anything, so it comes first; the result so far is `(pie)`. Remove `pie` from S and pairs mentioning `pie` from R to get $S = \{$`apple, cinnamon, fruit, spice, food, standard-object, t`$\}$ and $R = \{$`(apple, cinnamon), (apple, fruit), (cinnamon, spice), (fruit, food),`
`(spice, food), (food, standard-object), (standard-object, t)`$\}$.

The class `apple` is not preceded by anything, so it is next; the result is `(pie apple)`. Removing

apple and the relevant pairs results in $S = \{$cinnamon, fruit, spice, food, standard-object, t$\}$ and $R = \{$(cinnamon, spice), (fruit, food), (spice, food), (food, standard-object), (standard-object, t)$\}$.

The classes cinnamon and fruit are not preceded by anything, so the one with a direct subclass rightmost in the class precedence list computed so far goes next. The class apple is a direct subclass of fruit, and the class pie is a direct subclass of cinnamon. Because apple appears to the right of pie in the precedence list, fruit goes next, and the result so far is (pie apple fruit). $S = \{$cinnamon, spice, food, standard-object, t$\}$; $R = \{$(cinnamon, spice), (spice, food), (food, standard-object), (standard-object, t)$\}$.

The class cinnamon is next, giving the result so far as (pie apple fruit cinnamon). At this point $S = \{$spice, food, standard-object, t$\}$; $R = \{$(spice, food), (food, standard-object), (standard-object, t)$\}$.

The classes spice, food, standard-object, and t are added in that order, and the class precedence list is (pie apple fruit cinnamon spice food standard-object t).

It is possible to write a set of class definitions that cannot be ordered. For example:

```
(defclass new-class (fruit apple) ())

(defclass apple (fruit) ())
```

The class fruit must precede apple because the local ordering of superclasses must be preserved. The class apple must precede fruit because a class always precedes its own superclasses. When this situation occurs, an error is signaled when the system tries to compute the class precedence list.

The following might appear to be a conflicting set of definitions:

```
(defclass pie (apple cinnamon) ())

(defclass pastry (cinnamon apple) ())

(defclass apple () ())

(defclass cinnamon () ())
```

The class precedence list for pie is (pie apple cinnamon standard-object t).

The class precedence list for pastry is (pastry cinnamon apple standard-object t).

It is not a problem for apple to precede cinnamon in the ordering of the superclasses of pie but not in the ordering for pastry. However, it is not possible to build a new class that has both pie and pastry as superclasses.

Generic Functions and Methods

A ***generic function*** is a function whose behavior depends on the classes or identities of the arguments supplied to it. The ***methods*** define the class-specific behavior and operations of the generic function. The following sections describe generic functions and methods.

Introduction to Generic Functions

A generic function object contains a set of methods, a lambda-list, a method combination type, and other information.

Like an ordinary Lisp function, a generic function takes arguments, performs a series of operations, and perhaps returns useful values. An ordinary function has a single body of code that is always executed when the function is called. A generic function has a set of bodies of code of which a subset is selected for execution. The selected bodies of code and the manner of their combination are determined by the classes or identities of one or more of the arguments to the generic function and by its method combination type.

Ordinary functions and generic functions are called with identical syntax.

Generic functions are true functions that can be passed as arguments and used as the first argument to **funcall** and **apply**.

In Common Lisp, a name can be given to an ordinary function in one of two ways: a ***global*** name can be given to a function using the **defun** construct; a ***local*** name can be given using the **flet** or **labels** special forms. A generic function can be given a global name using the **defmethod** or **defgeneric** construct. A generic function can be given a local name using the **generic-flet**, **generic-labels**, or **with-added-methods** special forms. The name of a generic function, like the name of an ordinary function, can be either a symbol or a two-element list whose first element is **setf** and whose second element is a symbol. This is true for both local and global names.

The **generic-flet** special form creates new local generic functions using the set of methods specified by the method definitions in the **generic-flet** form. The scoping of generic function names within a **generic-flet** form is the same as for **flet**.

The **generic-labels** special form creates a set of new mutually recursive local generic functions using the set of methods specified by the method definitions in the **generic-labels** form. The scoping of generic function names within a **generic-labels** form is the same as for **labels**.

The **with-added-methods** special form creates new local generic functions by adding the set of methods specified by the method definitions with a given name in the **with-added-methods** form to copies of the methods of the lexically visible generic function of the same name. If there is a lexically visible ordinary function of the same name as one of specified generic functions, that function becomes the method function of the default method for the new generic function of that name.

The **generic-function** macro creates an anonymous generic function with the set of methods specified by the method definitions in the **generic-function** form.

When a **defgeneric** form is evaluated, one of three actions is taken:

- If a generic function of the given name already exists, the existing generic function object is modified. Methods specified by the current **defgeneric** form are added, and any methods in the existing generic function that were defined by a previous **defgeneric** form are removed. Methods added by the current **defgeneric** form might replace methods defined by **defmethod** or **defclass**. No other methods in the generic function are affected or replaced.
- If the given name names a non-generic function, a macro, or a special form, an error is signaled.
- Otherwise a generic function is created with the methods specified by the method definitions in the **defgeneric** form.

Some forms specify the options of a generic function, such as the type of method combination it uses or its argument precedence order. These forms will be referred to as "forms that specify generic function options." These forms are: **defgeneric**, **generic-function**, **generic-flet**, **generic-labels**, and **with-added-methods**.

Some forms define methods for a generic function. These forms will be referred to as "method-defining forms." These forms are: **defgeneric**, **defmethod**, **generic-function**, **generic-flet**, **generic-labels**, **with-added-methods**, and **defclass**. Note that all the method-defining forms except **defclass** and **defmethod** can specify generic function options and so are also forms that specify generic function options.

Introduction to Methods

A method object contains a method function, a sequence of ***parameter specializers*** that specify when the given method is applicable, a lambda-list, and a sequence of ***qualifiers*** that are used by the method combination facility to distinguish among methods.

A method object is not a function and cannot be invoked as a function. Various mechanisms in the Object System take a method object and invoke its method function, as is the case when a generic function is invoked. When this occurs it is said that the method is invoked or called.

A method-defining form contains the code that is to be run when the arguments to the generic function cause the method that it defines to be invoked. When a method-defining form is evaluated, a method object is created and one of four actions is taken:

- If a generic function of the given name already exists and if a method object already exists that agrees with the new one on parameter specializers and qualifiers, the new method object replaces the old one. For a definition of one method agreeing with another on parameter specializers and qualifiers, see the section "Agreement on Parameter Specializers and Qualifiers."

- If a generic function of the given name already exists and if there is no method object that agrees with the new one on parameter specializers and qualifiers, the existing generic function object is modified to contain the new method object.
- If the given name names a non-generic function, a macro, or a special form, an error is signaled.
- Otherwise a generic function is created with the methods specified by the method-defining form.

If the lambda-list of a new method is not congruent with the lambda-list of the generic function, an error is signaled. If a method-defining form that cannot specify generic function options creates a new generic function, a lambda-list for that generic function is derived from the lambda-lists of the methods in the method-defining form in such a way as to be congruent with them. For a discussion of ***congruence***, see the section "Congruent Lambda-lists for All Methods of a Generic Function."

Each method has a ***specialized lambda-list***, which determines when that method can be applied. A specialized lambda-list is like an ordinary lambda-list except that a ***specialized parameter*** may occur instead of the name of a required parameter. A specialized parameter is a list (*variable-name parameter-specializer-name*), where *parameter-specializer-name* is one of the following:

- A name that names a class
- (**eql** *form*)

A parameter specializer name denotes a parameter specializer as follows:

- A name that names a class denotes that class.
- The list (**eql** *form*) denotes (**eql** *object*), where *object* is the result of evaluating *form*. The form *form* is evaluated in the lexical environment in which the method-defining form is evaluated. Note that *form* is evaluated only once, at the time the method is defined, not each time the generic function is called.

Parameter specializer names are used in macros intended as the user-level interface (**defmethod**), while parameter specializers are used in the functional interface.

Only required parameters may be specialized, and there must be a parameter specializer for each required parameter. For notational simplicity, if some required parameter in a specialized lambda-list in a method-defining form is simply a variable name, its parameter specializer defaults to the class named **t**.

Given a generic function and a set of arguments, an ***applicable method*** is a method for that generic function whose parameter specializers are satisfied by their corresponding arguments. The following definition specifies what it means for a method to be applicable and for an argument to satisfy a parameter specializer.

Let $\langle A_1, \ldots, A_n \rangle$ be the required arguments to a generic function in order. Let $\langle P_1, \ldots, P_n \rangle$ be the parameter specializers corresponding to the required parameters of the method M in order. The method M is ***applicable*** when each A_i ***satisfies*** P_i. If P_i is a class, and if A_i is an instance of a class C, then it is said that A_i ***satisfies*** P_i when $C = P_i$ or when C is a subclass of P_i. If P_i is `(`**eql** *object*`)`, then it is said that A_i satisfies P_i when the function **eql** applied to A_i and *object* is true.

Because a parameter specializer is a type specifier, the function **typep** can be used during method selection to determine whether an argument satisfies a parameter specializer. In general a parameter specializer cannot be a type specifier list, such as `(`**vector single-float**`)`. The only parameter specializer that can be a list is `(`**eql** *object*`)`. This requires that Common Lisp be modified to include the type specifier **eql** to be defined as if the following were evaluated:

`(`**deftype eql** `(`*object*`) '(`**member** *,object*`))`

A method all of whose parameter specializers are the class named **t** is called a ***default method***; it is always applicable but may be shadowed by a more specific method.

Methods can have ***qualifiers***, which give the method combination procedure a way to distinguish among methods. A method that has one or more qualifiers is called a ***qualified*** method. A method with no qualifiers is called an ***unqualified method***. A qualifier is any object other than a list, that is, any non-**nil** atom. The qualifiers defined by standard method combination and by the built-in method combination types are symbols.

In this specification, the terms ***primary method*** and ***auxiliary method*** are used to partition methods within a method combination type according to their intended use. In standard method combination, primary methods are unqualified methods and auxiliary methods are methods with a single qualifier that is one of **:around**, **:before**, or **:after**. When a method combination type is defined using the short form of **define-method-combination**, primary methods are methods qualified with the name of the type of method combination, and auxiliary methods have the qualifier **:around**. Thus the terms ***primary method*** and ***auxiliary method*** have only a relative definition within a given method combination type.

Agreement on Parameter Specializers and Qualifiers

Two methods are said to agree with each other on parameter specializers and qualifiers if the following conditions hold:

1. Both methods have the same number of required parameters. Suppose the parameter specializers of the two methods are $P_{1,1} \ldots P_{1,n}$ and $P_{2,1} \ldots P_{2,n}$.

2. For each $1 \leq i \leq n$, $P_{1,i}$ agrees with $P_{2,i}$. The parameter specializer $P_{1,i}$ agrees with $P_{2,i}$ if $P_{1,i}$ and $P_{2,i}$ are the same class or if $P_{1,i} =$ `(`**eql** $object_1$`)`, $P_{2,i} =$ `(`**eql** $object_2$`)`, and `(`**eql** $object_1$ $object_2$`)`. Otherwise $P_{1,i}$ and $P_{2,i}$ do not agree.

3. The lists of qualifiers of both methods contain the same non-**nil** atoms in the same order. That is, the lists are **equal**.

Congruent Lambda-Lists for All Methods of a Generic Function

These rules define the congruence of a set of lambda-lists, including the lambda-list of each method for a given generic function and the lambda-list specified for the generic function itself, if given.

1. Each lambda-list must have the same number of required parameters.
2. Each lambda-list must have the same number of optional parameters. Each method can supply its own default for an optional parameter.
3. If any lambda-list mentions **&rest** or **&key**, each lambda-list must mention one or both of them.
4. If the generic function lambda-list mentions **&key**, each method must accept all of the keyword names mentioned after **&key**, either by accepting them explicitly, by specifying **&allow-other-keys**, or by specifying **&rest** but not **&key**. Each method can accept additional keyword arguments of its own. The checking of the validity of keyword names is done in the generic function, not in each method. A method is invoked as if the keyword argument pair whose keyword is **:allow-other-keys** and whose value is **t** were supplied, though no such argument pair will be passed.
5. The use of **&allow-other-keys** need not be consistent across lambda-lists. If **&allow-other-keys** is mentioned in the lambda-list of any applicable method or of the generic function, any keyword arguments may be mentioned in the call to the generic function.
6. The use of **&aux** need not be consistent across methods.

   If a method-defining form that cannot specify generic function options creates a generic function, and if the lambda-list for the method mentions keyword arguments, the lambda-list of the generic function will mention **&key** (but no keyword arguments).

Keyword Arguments in Generic Functions and Methods

When a generic function or any of its methods mentions **&key** in a lambda-list, the specific set of keyword arguments accepted by the generic function varies according to the applicable methods. The set of keyword arguments accepted by the generic function for a particular call is the union of the keyword arguments accepted by all applicable methods and the keyword arguments mentioned after **&key** in the generic function definition, if any. A method that has **&rest** but not **&key** does not affect the set of acceptable keyword arguments. If the lambda-list of any applicable method or of the generic function definition contains **&allow-other-keys**, all keyword arguments are accepted by the generic function.

The lambda-list congruence rules require that each method accept all of the keyword arguments mentioned after **&key** in the generic function definition, by accepting them explicitly, by specifying **&allow-other-keys**, or by specifying **&rest** but not **&key**. Each method can accept additional keyword arguments of its own, in addition to the keyword arguments mentioned in the generic function definition.

If a generic function is passed a keyword argument that no applicable method accepts, an error is signaled.

For example, suppose there are two methods defined for `width` as follows:

```
(defmethod width ((c character-class) &key font) ...)

(defmethod width ((p picture-class) &key pixel-size) ...)
```

Assume that there are no other methods and no generic function definition for `width`. The evaluation of the following form will signal an error because the keyword argument `:pixel-size` is not accepted by the applicable method.

```
(width (make-instance ‘character-class :char #\Q)
       :font ’baskerville :pixel-size 10)
```

The evaluation of the following form will signal an error.

```
(width (make-instance ‘picture-class :glyph (glyph #\Q))
       :font ’baskerville :pixel-size 10)
```

The evaluation of the following form will not signal an error if the class named `character-picture-class` is a subclass of both `picture-class` and `character-class`.

```
(width (make-instance ‘character-picture-class :char #\Q)
       :font ’baskerville :pixel-size 10)
```

Method Selection and Combination

When a generic function is called with particular arguments, it must determine the code to execute. This code is called the ***effective method*** for those arguments. The effective method is a ***combination*** of the applicable methods in the generic function. A combination of methods is a Lisp expression that contains calls to some or all of the methods. If a generic function is called and no methods apply, the generic function **no-applicable-method** is invoked.

When the effective method has been determined, it is invoked with the same arguments that were passed to the generic function. Whatever values it returns are returned as the values of the generic function.

Determining the Effective Method

The effective method is determined by the following three-step procedure:

1. Select the applicable methods.
2. Sort the applicable methods by precedence order, putting the most specific method first.
3. Apply method combination to the sorted list of applicable methods, producing the effective method.

Selecting the Applicable Methods

This step is described in the section "Introduction to Methods."

Sorting the Applicable Methods by Precedence Order

To compare the precedence of two methods, their parameter specializers are examined in order. The default examination order is from left to right, but an alternative order may be specified by the **:argument-precedence-order** option to **defgeneric** or to any of the other forms that specify generic function options.

The corresponding parameter specializers from each method are compared. When a pair of parameter specializers are equal, the next pair are compared for equality. If all corresponding parameter specializers are equal, the two methods must have different qualifiers; in this case, either method can be selected to precede the other.

If some corresponding parameter specializers are not equal, the first pair of parameter specializers that are not equal determines the precedence. If both parameter specializers are classes, the more specific of the two methods is the method whose parameter specializer appears earlier in the class precedence list of the corresponding argument. Because of the way in which the set of applicable methods is chosen, the parameter specializers are guaranteed to be present in the class precedence list of the class of the argument.

If just one parameter specializer is (**eql** *object*), the method with that parameter specializer precedes the other method. If both parameter specializers are **eql** forms, the specializers must be the same (otherwise the two methods would not both have been applicable to this argument).

The resulting list of applicable methods has the most specific method first and the least specific method last.

Applying Method Combination to the Sorted List of Applicable Methods

In the simple case—if standard method combination is used and all applicable methods are primary methods—the effective method is the most specific method. That method can call the next most specific method by using the function **call-next-method**. The method that **call-next-method** will call is referred to as the ***next method***. The predicate **next-method-p** tests whether a next method exists. If **call-next-method** is called and there is no next most specific method, the generic function **no-next-method** is invoked.

In general, the effective method is some combination of the applicable methods. It is defined by a Lisp form that contains calls to some or all of the applicable methods, returns the value or values that will be returned as the value or values of the generic function, and optionally makes some of the methods accessible by means of **call-next-method**. This Lisp form is the body of the effective method; it is augmented with an appropriate lambda-list to make it a function.

The role of each method in the effective method is determined by its method qualifiers and the specificity of the method. A qualifier serves to mark a method, and the meaning of a qualifier is determined by the way that these marks are used by this step of the procedure. If an applicable method has an unrecognized qualifier, this step signals an error and does not include that method in the effective method.

When standard method combination is used together with qualified methods, the effective method is produced as described in the section "Standard Method Combination."

Another type of method combination can be specified by using the **:method-combination** option of **defgeneric** or of any of the other forms that specify generic function options. In this way this step of the procedure can be customized.

New types of method combination can be defined by using the **define-method-combination** macro.

The meta-object level also offers a mechanism for defining new types of method combination. The generic function **compute-effective-method** receives as arguments the generic function, the method combination object, and the sorted list of applicable methods. It returns the Lisp form that defines the effective method. A method for **compute-effective-method** can be defined directly by using **defmethod** or indirectly by using **define-method-combination**. A ***method combination object*** is an object that encapsulates the method combination type and options specified by the **:method-combination** option to forms that specify generic function options.

Implementation Note:
In the simplest implementation, the generic function would compute the effective method each time it was called. In practice, this will be too inefficient for some implementations. Instead, these implementations might employ a variety of optimizations of the three-step procedure. Some illustrative examples of such optimizations are the following:

- Use a hash table keyed by the class of the arguments to store the effective method.
- Compile the effective method and save the resulting compiled function in a table.
- Recognize the Lisp form as an instance of a pattern of control structure and substitute a closure that implements that structure.
- Examine the parameter specializers of all methods for the generic function and enumerate all possible effective methods. Combine the effective methods, together with code to select from among them, into a single function and compile that function. Call that function whenever the generic function is called.

Standard Method Combination

Standard method combination is supported by the class **standard-generic-function**. It is used if no other type of method combination is specified or if the built-in method combination type **standard** is specified.

Primary methods define the main action of the effective method, while ***auxiliary methods*** modify that action in one of three ways. A primary method has no method qualifiers.

An auxiliary method is a method whose method qualifier is **:before**, **:after**, or **:around**. Standard method combination allows no more than one qualifier per method; if a method definition specifies more than one qualifier per method, an error is signaled.

- A **:before** method has the keyword **:before** as its only qualifier. A **:before** method specifies code that is to be run before any primary methods.
- An **:after** method has the keyword **:after** as its only qualifier. An **:after** method specifies code that is to be run after primary methods.
- An **:around** method has the keyword **:around** as its only qualifier. An **:around** method specifies code that is to be run instead of other applicable methods but which is able to cause some of them to be run.

The semantics of standard method combination is as follows:

- If there are any **:around** methods, the most specific **:around** method is called. It supplies the value or values of the generic function.
- Inside the body of an **:around** method, **call-next-method** can be used to call the next method. When the next method returns, the **:around** method can execute more code, perhaps based on the returned value or values. The generic function **no-next-method** is invoked if **call-next-method** is used and there is no applicable method to call. The function **next-method-p** may be used to determine whether a next method exists.
- If an **:around** method invokes **call-next-method**, the next most specific **:around** method is called, if one is applicable. If there are no **:around** methods or if **call-next-method** is called by the least specific **:around** method, the other methods are called as follows:
  - All the **:before** methods are called, in most-specific-first order. Their values are ignored. An error is signaled if **call-next-method** is used in a **:before** method.
  - The most specific primary method is called. Inside the body of a primary method, **call-next-method** may be used to call the next most specific primary method. When that method returns, the previous primary method can execute more code, perhaps based on the returned value or values. The generic function **no-next-method** is invoked if **call-next-method** is used and there are no more applicable primary methods. The function **next-method-p** may be used to determine whether a next method exists. If **call-next-method** is not used, only the most specific primary method is called.
  - All the **:after** methods are called in most-specific-last order. Their values are ignored. An error is signaled if **call-next-method** is used in an **:after** method.
- If no **:around** methods were invoked, the most specific primary method supplies the value or values returned by the generic function. The value or values returned by the invocation of **call-next-method** in the least specific **:around** method are those returned by the most specific primary method.

In standard method combination, if there is an applicable method but no applicable primary method, an error is signaled.

The **:before** methods are run in most-specific-first order while the **:after** methods are run in least-specific-first order. The design rationale for this difference can be illustrated with an example. Suppose class C_1 modifies the behavior of its superclass, C_2, by adding **:before** and **:after** methods. Whether the behavior of the class C_2 is defined directly by methods on C_2 or is inherited from its superclasses does not affect the relative order of invocation of methods on instances of the class C_1. Class C_1's **:before** method runs before all of class C_2's methods. Class C_1's **:after** method runs after all of class C_2's methods.

By contrast, all **:around** methods run before any other methods run. Thus a less specific **:around** method runs before a more specific primary method.

If only primary methods are used and if **call-next-method** is not used, only the most specific method is invoked; that is, more specific methods shadow more general ones.

Declarative Method Combination

The macro **define-method-combination** defines new forms of method combination. It provides a mechanism for customizing the production of the effective method. The default procedure for producing an effective method is described in the section "Determining the Effective Method." There are two forms of **define-method-combination**. The short form is a simple facility while the long form is more powerful and more verbose. The long form resembles **defmacro** in that the body is an expression that computes a Lisp form; it provides mechanisms for implementing arbitrary control structures within method combination and for arbitrary processing of method qualifiers. The syntax and use of both forms of **define-method-combination** are explained in Chapter 2.

Built-in Method Combination Types

The Common Lisp Object System provides a set of built-in method combination types. To specify that a generic function is to use one of these method combination types, the name of the method combination type is given as the argument to the **:method-combination** option to **defgeneric** or to the **:method-combination** option to any of the other forms that specify generic function options.

The names of the built-in method combination types are +, **and**, **append**, **list**, **max**, **min**, **nconc**, **or**, **progn**, and **standard**.

The semantics of the **standard** built-in method combination type was described in the section "Standard Method Combination." The other built-in method combination types are called ***simple built-in method combination types.***

The simple built-in method combination types act as though they were defined by the short form of **define-method-combination**. They recognize two roles for methods:

- An **:around** method has the keyword symbol **:around** as its sole qualifier. The meaning of **:around** methods is the same as in standard method combination. Use of the functions **call-next-method** and **next-method-p** is supported in **:around** methods.
- A primary method has the name of the method combination type as its sole qualifier. For example, the built-in method combination type **and** recognizes methods whose sole qualifier is **and**; these are primary methods. Use of the functions **call-next-method** and **next-method-p** is not supported in primary methods.

The semantics of the simple built-in method combination types is as follows:

- If there are any **:around** methods, the most specific **:around** method is called. It supplies the value or values of the generic function.

- Inside the body of an **:around** method, the function **call-next-method** can be used to call the next method. The generic function **no-next-method** is invoked if **call-next-method** is used and there is no applicable method to call. The function **next-method-p** may be used to determine whether a next method exists. When the next method returns, the **:around** method can execute more code, perhaps based on the returned value or values.

- If an **:around** method invokes **call-next-method**, the next most specific **:around** method is called, if one is applicable. If there are no **:around** methods or if **call-next-method** is called by the least specific **:around** method, a Lisp form derived from the name of the built-in method combination type and from the list of applicable primary methods is evaluated to produce the value of the generic function. Suppose the name of the method combination type is *operator* and the call to the generic function is of the form

  $$(\textit{generic-function}\ a_1 \ldots a_n)$$

  Let $M_1, \ldots, M_k$ be the applicable primary methods in order; then the derived Lisp form is

  $$(\textit{operator}\ \langle M_1\ a_1 \ldots a_n\rangle \ldots \langle M_k\ a_1 \ldots a_n\rangle)$$

  If the expression $\langle M_i\ a_1 \ldots a_n\rangle$ is evaluated, the method M_i will be applied to the arguments $a_1 \ldots a_n$. For example, if *operator* is **or**, the expression $\langle M_i\ a_1 \ldots a_n\rangle$ is evaluated only if $\langle M_j\ a_1 \ldots a_n\rangle$, $1 \leq j < i$, returned **nil**.

  The default order for the primary methods is **:most-specific-first**. However, the order can be reversed by supplying **:most-specific-last** as the second argument to the **:method-combination** option.

The simple built-in method combination types require exactly one qualifier per method. An error is signaled if there are applicable methods with no qualifiers or with qualifiers that are not supported by the method combination type. An error is signaled if there are applicable **:around** methods and no applicable primary methods.

Meta-Objects

The implementation of the Object System manipulates classes, methods, and generic functions. The meta-object protocol specifies a set of generic functions defined by methods on classes; the behavior of those generic functions defines the behavior of the Object System. The instances of the classes on which those methods are defined are called ***meta-objects***. Programming at the meta-object protocol level involves defining new classes of meta-objects along with methods specialized on these classes.

Metaclasses

The ***metaclass*** of an object is the class of its class. The metaclass determines the representation of instances of its instances and the forms of inheritance used by its instances for slot descriptions and method inheritance. The metaclass mechanism can be used to provide particular forms of optimization or to tailor the Common Lisp Object System for particular uses. The protocol for defining metaclasses is discussed in the chapter "The Common Lisp Object System Meta-Object Protocol."

Standard Metaclasses

The Common Lisp Object System provides a number of predefined metaclasses. These include the classes **standard-class**, **built-in-class**, and **structure-class**:

- The class **standard-class** is the default class of classes defined by **defclass**.
- The class **built-in-class** is the class whose instances are classes that have special implementations with restricted capabilities. Any class that corresponds to a standard Common Lisp type specified in *Common Lisp: The Language* might be an instance of **built-in-class**. The predefined Common Lisp type specifiers that are required to have corresponding classes are listed in Figure 1-1. It is implementation dependent whether each of these classes is implemented as a built-in class.
- All classes defined by means of **defstruct** are instances of **structure-class**.

Standard Meta-objects

The Object System supplies a set of meta-objects, called ***standard meta-objects***. These include the class **standard-object** and instances of the classes **standard-method**, **standard-generic-function**, and **method-combination**.

- The class **standard-method** is the default class of methods defined by the forms **defmethod**, **defgeneric**, **generic-function**, **generic-flet**, **generic-labels**, and **with-added-methods**.

- The class **standard-generic-function** is the default class of generic functions defined by the forms **defmethod**, **defgeneric**, **generic-function**, **generic-flet**, **generic-labels**, **with-added-methods**, and **defclass**.
- The class named **standard-object** is an instance of the class **standard-class** and is a superclass of every class that is an instance of **standard-class** except itself and **structure-class**.
- Every method combination object is an instance of a subclass of the class **method-combination**.

Object Creation and Initialization

The generic function **make-instance** creates and returns a new instance of a class. The first argument is a class or the name of a class, and the remaining arguments form an ***initialization argument*** list.

The initialization of a new instance consists of several distinct steps, including the following: combining the explicitly supplied initialization arguments with default values for the unsupplied initialization arguments, checking the validity of the initialization arguments, allocating storage for the instance, filling slots with values, and executing user-supplied methods that perform additional initialization. Each step of **make-instance** is implemented by a generic function to provide a mechanism for customizing that step. In addition, **make-instance** is itself a generic function and thus also can be customized.

The Object System specifies system-supplied primary methods for each step and thus specifies a well-defined standard behavior for the entire initialization process. The standard behavior provides four simple mechanisms for controlling initialization:

- Declaring a symbol to be an initialization argument for a slot. An initialization argument is declared by using the **:initarg** slot option to **defclass**. This provides a mechanism for supplying a value for a slot in a call to **make-instance**.
- Supplying a default value form for an initialization argument. Default value forms for initialization arguments are defined by using the **:default-initargs** class option to **defclass**. If an initialization argument is not explicitly provided as an argument to **make-instance**, the default value form is evaluated in the lexical environment of the **defclass** form that defined it, and the resulting value is used as the value of the initialization argument.
- Supplying a default initial value form for a slot. A default initial value form for a slot is defined by using the **:initform** slot option to **defclass**. If no initialization argument associated with that slot is given as an argument to **make-instance** or is defaulted by **:default-initargs**, this default initial value form is evaluated in the lexical environment of the **defclass** form that defined it, and the resulting value is stored in the slot. The **:initform** form for a local slot may be used when creating an instance, when updating an instance to conform to a redefined class, or when updating an instance to conform to the definition of a different class. The **:initform** form for a shared slot may be used when defining or re-defining the class.
- Defining methods for **initialize-instance** and **shared-initialize**. The slot-filling behavior described above is implemented by a system-supplied primary method for **initialize-instance** which invokes **shared-initialize**. The generic function **shared-initialize** implements the parts of initialization shared by these four situations: when making an instance, when re-initializing an instance, when updating an instance to conform to a redefined class, and when updating an instance to conform to the definition of a different class. The system-supplied primary method for **shared-initialize** directly implements the slot-filling behavior described above, and **initialize-instance** simply invokes **shared-initialize**.

Initialization Arguments

An initialization argument controls object creation and initialization. It is often convenient to use keyword symbols to name initialization arguments, but the name of an initialization argument can be any symbol, including **nil**. An initialization argument can be used in two ways: to fill a slot with a value or to provide an argument for an initialization method. A single initialization argument can be used for both purposes.

An ***initialization argument list*** is a list of alternating initialization argument names and values. Its structure is identical to a property list and also to the portion of an argument list processed for **&key** parameters. As in those lists, if an initialization argument name appears more than once in an initialization argument list, the leftmost occurrence supplies the value and the remaining occurrences are ignored. The arguments to **make-instance** (after the first argument) form an initialization argument list. Error-checking of initialization argument names is disabled if the keyword argument pair whose keyword is **:allow-other-keys** and whose value is non-**nil** appears in the initialization argument list.

An initialization argument can be associated with a slot. If the initialization argument has a value in the initialization argument list, the value is stored into the slot of the newly created object, overriding any **:initform** form associated with the slot. A single initialization argument can initialize more than one slot. An initialization argument that initializes a shared slot stores its value into the shared slot, replacing any previous value.

An initialization argument can be associated with a method. When an object is created and a particular initialization argument is supplied, the generic functions **initialize-instance**, **shared-initialize**, and **allocate-instance** are called with that initialization argument's name and value as a keyword argument pair. If a value for the initialization argument is not supplied in the initialization argument list, the method's lambda-list supplies a default value.

Initialization arguments are used in four situations: when making an instance, when re-initializing an instance, when updating an instance to conform to a redefined class, and when updating an instance to conform to the definition of a different class.

Because initialization arguments are used to control the creation and initialization of an instance of some particular class, we say that an initialization argument is "an initialization argument for" that class.

Declaring the Validity of Initialization Arguments

Initialization arguments are checked for validity in each of the four situations that use them. An initialization argument may be valid in one situation and not another. For example, the system-supplied primary method for **make-instance** defined for the class **standard-class** checks the validity of its initialization arguments and signals an error if an initialization argument is supplied that is not declared as valid in that situation.

There are two means for declaring initialization arguments valid.

- Initialization arguments that fill slots are declared as valid by the **:initarg** slot option to **defclass**. The **:initarg** slot option is inherited from superclasses. Thus the set of valid initialization arguments that fill slots for a class is the union of the initialization arguments that fill slots declared as valid by that class and its superclasses. Initialization arguments that fill slots are valid in all four contexts.
- Initialization arguments that supply arguments to methods are declared as valid by defining those methods. The keyword name of each keyword parameter specified in the method's lambda-list becomes an initialization argument for all classes for which the method is applicable. Thus method inheritance controls the set of valid initialization arguments that supply arguments to methods. The generic functions for which method definitions serve to declare initialization arguments valid are as follows:
  - Making an instance of a class: **allocate-instance**, **initialize-instance**, and **shared-initialize**. Initialization arguments declared as valid by these methods are valid when making an instance of a class.
  - Re-initializing an instance: **reinitialize-instance** and **shared-initialize**. Initialization arguments declared as valid by these methods are valid when re-initializing an instance.
  - Updating an instance to conform to a redefined class: **update-instance-for-redefined-class** and **shared-initialize**. Initialization arguments declared as valid by these methods are valid when updating an instance to conform to a redefined class.
  - Updating an instance to conform to the definition of a different class: **update-instance-for-different-class** and **shared-initialize**. Initialization arguments declared as valid by these methods are valid when updating an instance to conform to the definition of a different class.

The set of valid initialization arguments for a class is the set of valid initialization arguments that either fill slots or supply arguments to methods, along with the predefined initialization argument **:allow-other-keys**. The default value for **:allow-other-keys** is **nil**. The meaning of **:allow-other-keys** is the same as when it is passed to an ordinary function.

Defaulting of Initialization Arguments

A ***default value form*** can be supplied for an initialization argument by using the **:default-initargs** class option. If an initialization argument is declared valid by some particular class, its default value form might be specified by a different class. In this case **:default-initargs** is used to supply a default value for an inherited initialization argument.

The **:default-initargs** option is used only to provide default values for initialization arguments; it does not declare a symbol as a valid initialization argument name. Furthermore, the **:default-initargs** option is used only to provide default values for initialization arguments when making an instance.

The argument to the **:default-initargs** class option is a list of alternating initialization argument names and forms. Each form is the default value form for the corresponding initialization argument. The default value form of an initialization argument is used and evaluated only if that initialization argument does not appear in the arguments to **make-instance** and is not defaulted by a more specific class. The default value form is evaluated in the lexical environment of the **defclass** form that supplied it; the resulting value is used as the initialization argument's value.

The initialization arguments supplied to **make-instance** are combined with defaulted initialization arguments to produce a ***defaulted initialization argument list***. A defaulted initialization argument list is a list of alternating initialization argument names and values in which unsupplied initialization arguments are defaulted and in which the explicitly supplied initialization arguments appear earlier in the list than the defaulted initialization arguments. Defaulted initialization arguments are ordered according to the order in the class precedence list of the classes that supplied the default values.

There is a distinction between the purposes of the **:default-initargs** and the **:initform** options with respect to the initialization of slots. The **:default-initargs** class option provides a mechanism for the user to give a default value form for an initialization argument without knowing whether the initialization argument initializes a slot or is passed to a method. If that initialization argument is not explicitly supplied in a call to **make-instance**, the default value form is used, just as if it had been supplied in the call. In contrast, the **:initform** slot option provides a mechanism for the user to give a default initial value form for a slot. An **:initform** form is used to initialize a slot only if no initialization argument associated with that slot is given as an argument to **make-instance** or is defaulted by **:default-initargs**.

The order of evaluation of default value forms for initialization arguments and the order of evaluation of **:initform** forms are undefined. If the order of evaluation is important, **initialize-instance** or **shared-initialize** methods should be used instead.

Rules for Initialization Arguments

The **:initarg** slot option may be specified more than once for a given slot.

The following rules specify when initialization arguments may be multiply defined:

- A given initialization argument can be used to initialize more than one slot if the same initialization argument name appears in more than one **:initarg** slot option.
- A given initialization argument name can appear in the lambda-list of more than one initialization method.
- A given initialization argument name can appear both in an **:initarg** slot option and in the lambda-list of an initialization method.

If two or more initialization arguments that initialize the same slot are given in the arguments to **make-instance**, the leftmost of these initialization arguments in the initialization argument list supplies the value, even if the initialization arguments have different names.

If two or more different initialization arguments that initialize the same slot have default values and none is given explicitly in the arguments to **make-instance**, the initialization argument that appears in a **:default-initargs** class option in the most specific of the classes supplies the value. If a single **:default-initargs** class option specifies two or more initialization arguments that initialize the same slot and none is given explicitly in the arguments to **make-instance**, the leftmost in the **:default-initargs** class option supplies the value, and the values of the remaining default value forms are ignored.

Initialization arguments given explicitly in the arguments to **make-instance** appear to the left of defaulted initialization arguments. Suppose that the classes C_1 and C_2 supply the values of defaulted initialization arguments for different slots, and suppose that C_1 is more specific than C_2; then the defaulted initialization argument whose value is supplied by C_1 is to the left of the defaulted initialization argument whose value is supplied by C_2 in the defaulted initialization argument list. If a single **:default-initargs** class option supplies the values of initialization arguments for two different slots, the initialization argument whose value is specified farther to the left in the **default-initargs** class option appears farther to the left in the defaulted initialization argument list.

If a slot has both an **:initform** form and an **:initarg** slot option, and the initialization argument is defaulted using **:default-initargs** or is supplied to **make-instance**, the captured **:initform** form is neither used nor evaluated.

The following is an example of the above rules:

```
(defclass q () ((x :initarg a)))
(defclass r (q) ((x :initarg b))
  (:default-initargs a 1 b 2))
```

Form	Defaulted Initialization Argument List	Contents of Slot
`(make-instance 'r)`	`(a 1 b 2)`	`1`
`(make-instance 'r 'a 3)`	`(a 3 b 2)`	`3`
`(make-instance 'r 'b 4)`	`(b 4 a 1)`	`4`
`(make-instance 'r 'a 1 'a 2)`	`(a 1 a 2 b 2)`	`1`

Shared-Initialize

The generic function **shared-initialize** is used to fill the slots of an instance using initialization arguments and **:initform** forms when an instance is created, when an instance is re-initialized, when an instance is updated to conform to a redefined class, and when an instance is updated to conform to a different class. It uses standard method combination. It takes the following arguments: the instance to be initialized, a specification of a set of names of slots accessible in

that instance, and any number of initialization arguments. The arguments after the first two must form an initialization argument list.

The second argument to **shared-initialize** may be one of the following:

- It can be list of slot names, which specifies the set of those slot names.
- It can be **nil**, which specifies the empty set of slot names.
- It can be the symbol **t**, which specifies the set of all of the slots.

There is a system-supplied primary method for **shared-initialize** whose first parameter specializer is the class **standard-object**. This method behaves as follows on each slot, whether shared or local:

- If an initialization argument in the initialization argument list specifies a value for that slot, that value is stored into the slot, even if a value has already been stored in the slot before the method is run. The affected slots are independent of which slots are indicated by the second argument to **shared-initialize**.
- Any slots indicated by the second argument that are still unbound at this point are initialized according to their **:initform** forms. For any such slot that has an **:initform** form, that form is evaluated in the lexical environment of its defining **defclass** form and the result is stored into the slot. For example, if a **:before** method stores a value in the slot, the **:initform** form will not be used to supply a value for the slot. If the second argument specifies a name that does not correspond to any slots accessible in the instance, the results are unspecified.
- The rules mentioned in the section "Rules for Initialization Arguments" are obeyed.

The generic function **shared-initialize** is called by the system-supplied primary methods for **reinitialize-instance**, **update-instance-for-different-class**, **update-instance-for-redefined-class**, and **initialize-instance**. Thus, methods can be written for **shared-initialize** to specify actions that should be taken in all of these contexts.

Initialize-Instance

The generic function **initialize-instance** is called by **make-instance** to initialize a newly created instance. It uses standard method combination. Methods for **initialize-instance** can be defined in order to perform any initialization that cannot be achieved with the simple slot-filling mechanisms.

During initialization, **initialize-instance** is invoked after the following actions have been taken:

- The defaulted initialization argument list has been computed by combining the supplied initialization argument list with any default initialization arguments for the class.
- The validity of the defaulted initialization argument list has been checked. If any of the initialization arguments has not been declared as valid, an error is signaled.

- A new instance whose slots are unbound has been created.

The generic function **initialize-instance** is called with the new instance and the defaulted initialization arguments. There is a system-supplied primary method for **initialize-instance** whose parameter specializer is the class **standard-object**. This method calls the generic function **shared-initialize** to fill in the slots according to the initialization arguments and the **:initform** forms for the slots; the generic function **shared-initialize** is called with the following arguments: the instance, **t**, and the defaulted initialization arguments.

Note that **initialize-instance** provides the defaulted initialization argument list in its call to **shared-initialize**, so the first step performed by the system-supplied primary method for **shared-initialize** takes into account both the initialization arguments provided in the call to **make-instance** and the defaulted initialization argument list.

Methods for **initialize-instance** can be defined to specify actions to be taken when an instance is initialized. If only **:after** methods for **initialize-instance** are defined, they will be run after the system-supplied primary method for initialization and therefore will not interfere with the default behavior of **initialize-instance**.

The Object System provides two functions that are useful in the bodies of **initialize-instance** methods. The function **slot-boundp** returns a boolean value that indicates whether a specified slot has a value; this provides a mechanism for writing **:after** methods for **initialize-instance** that initialize slots only if they have not already been initialized. The function **slot-makunbound** causes the slot to have no value.

Definitions of Make-Instance and Initialize-Instance

The generic function **make-instance** behaves as if it were defined as follows, except that certain optimizations are permitted:

```
(defmethod make-instance ((class standard-class) &rest initargs)
  (setq initargs (default-initargs class initargs))
  ...
  (let ((instance (apply #'allocate-instance class initargs)))
    (apply #'initialize-instance instance initargs)
    instance))

(defmethod make-instance ((class-name symbol) &rest initargs)
  (apply #'make-instance (find-class class-name) initargs))
```

The elided code in the definition of **make-instance** checks the supplied initialization arguments to determine whether an initialization argument was supplied that neither filled a slot nor supplied an argument to an applicable method. This check could be implemented using the generic functions **class-prototype**, **compute-applicable-methods**, **function-keywords**, and **class-slot-initargs**. See Chapter 3 for a description of this initialization argument check.

The generic function **initialize-instance** behaves as if it were defined as follows, except that certain optimizations are permitted:

```
(defmethod initialize-instance ((instance standard-object) &rest initargs)
  (apply #'shared-initialize instance t initargs)))
```

These procedures can be customized at either the Programmer Interface level, the meta-object level, or both.

Customizing at the Programmer Interface level includes using the **:initform**, **:initarg**, and **:default-initargs** options to **defclass**, as well as defining methods for **make-instance** and **initialize-instance**. It is also possible to define methods for **shared-initialize**, which would be invoked by the generic functions **reinitialize-instance**, **update-instance-for-redefined-class**, **update-instance-for-different-class**, and **initialize-instance**. The meta-object level supports additional customization by allowing methods to be defined on **make-instance**, **default-initargs**, and **allocate-instance**. Chapters 2 and 3 document each of these generic functions and the system-supplied primary methods.

Implementations are permitted to make certain optimizations to **initialize-instance** and **shared-initialize**. The description of **shared-initialize** in Chapter 2 mentions the possible optimizations.

Because of optimization, the check for valid initialization arguments might not be implemented using the generic functions **class-prototype**, **compute-applicable-methods**, **function-keywords**, and **class-slot-initargs**. In addition, methods for the generic function **default-initargs**, and the system-supplied primary methods for **allocate-instance**, **initialize-instance**, and **shared-initialize** might not be called on every call to **make-instance** or might not receive exactly the arguments that would be expected.

Redefining Classes

A class that is an instance of **standard-class** can be redefined if the new class will also be an instance of **standard-class**. Redefining a class modifies the existing class object to reflect the new class definition; it does not create a new class object for the class. Any method object created by a **:reader**, **:writer**, or **:accessor** option specified by the old **defclass** form is removed from the corresponding generic function. Methods specified by the new **defclass** form are added.

When the class *C* is redefined, changes are propagated to its instances and to instances of any of its subclasses. Updating such an instance occurs at an implementation-dependent time, but no later than the next time a slot of that instance is read or written. Updating an instance does not change its identity as defined by the **eq** function. The updating process may change the slots of that particular instance, but it does not create a new instance. Whether updating an instance consumes storage is implementation dependent.

Note that redefining a class may cause slots to be added or deleted. If a class is redefined in a way that changes the set of local slots accessible in instances, the instances will be updated. It is implementation dependent whether instances are updated if a class is redefined in a way that does not change the set of local slots accessible in instances.

The value of a slot that is specified as shared both in the old class and in the new class is retained. If such a shared slot was unbound in the old class, it will be unbound in the new class. Slots that were local in the old class and that are shared in the new class are initialized. Newly added shared slots are initialized.

Each newly added shared slot is set to the result of evaluating the captured **:initform** form for the slot that was specified in the **defclass** form for the new class. If there is no **:initform** form, the slot is unbound.

If a class is redefined in such a way that the set of local slots accessible in an instance of the class is changed, a two-step process of updating the instances of the class takes place. The process may be explicitly started by invoking the generic function **make-instances-obsolete**. This two-step process can happen in other circumstances in some implementations. For example, in some implementations this two-step process will be triggered if the order of slots in storage is changed.

The first step modifies the structure of the instance by adding new local slots and discarding local slots that are not defined in the new version of the class. The second step initializes the newly added local slots and performs any other user-defined actions. These two steps are further specified in the next two sections.

Modifying the Structure of Instances

The first step modifies the structure of instances of the redefined class to conform to its new class definition. Local slots specified by the new class definition that are not specified as either local or shared by the old class are added, and slots not specified as either local or shared by the

new class definition that are specified as local by the old class are discarded. The names of these added and discarded slots are passed as arguments to **update-instance-for-redefined-class** as described in the next section.

The values of local slots specified by both the new and old classes are retained. If such a local slot was unbound, it remains unbound.

The value of a slot that is specified as shared in the old class and as local in the new class is retained. If such a shared slot was unbound, the local slot will be unbound.

Initializing Newly Added Local Slots

The second step initializes the newly added local slots and performs any other user-defined actions. This step is implemented by the generic function **update-instance-for-redefined-class**, which is called after completion of the first step of modifying the structure of the instance.

The generic function **update-instance-for-redefined-class** takes four required arguments: the instance being updated after it has undergone the first step, a list of the names of local slots that were added, a list of the names of local slots that were discarded, and a property list containing the slot names and values of slots that were discarded and had values. Included among the discarded slots are slots that were local in the old class and that are shared in the new class.

The generic function **update-instance-for-redefined-class** also takes any number of initialization arguments. When it is called by the system to update an instance whose class has been redefined, no initialization arguments are provided.

There is a system-supplied primary method for **update-instance-for-redefined-class** whose parameter specializer for its instance argument is the class **standard-object**. First this method checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared as valid. (See the section "Declaring the Validity of Initialization Arguments" for more information.) Then it calls the generic function **shared-initialize** with the following arguments: the instance, the list of names of the newly added slots, and the initialization arguments it received.

Customizing Class Redefinition

Methods for **update-instance-for-redefined-class** may be defined to specify actions to be taken when an instance is updated. If only **:after** methods for **update-instance-for-redefined-class** are defined, they will be run after the system-supplied primary method for initialization and therefore will not interfere with the default behavior of **update-instance-for-redefined-class**. Because no initialization arguments are passed to **update-instance-for-redefined-class** when it is called by the system, the **:initform** forms for slots that are filled by **:before** methods for **update-instance-for-redefined-class** will not be evaluated by **shared-initialize**.

Methods for **shared-initialize** may be defined to customize class redefinition. See the section "Shared-Initialize" for more information.

Extensions

There are two allowed extensions to class redefinition:

- The Object System may be extended to permit the new class to be an instance of a metaclass other than the metaclass of the old class.
- The Object System may be extended to support an updating process when either the old or the new class is an instance of a class other than **standard-class** that is not a built-in class.

Changing the Class of an Instance

The function **change-class** can be used to change the class of an instance from its current class, C_{from}, to a different class, C_{to}; it changes the structure of the instance to conform to the definition of the class C_{to}.

Note that changing the class of an instance may cause slots to be added or deleted.

When **change-class** is invoked on an instance, a two-step updating process takes place. The first step modifies the structure of the instance by adding new local slots and discarding local slots that are not specified in the new version of the instance. The second step initializes the newly added local slots and performs any other user-defined actions. These two steps are further described in the two following sections.

Modifying the Structure of the Instance

In order to make the instance conform to the class C_{to}, local slots specified by the class C_{to} that are not specified by the class C_{from} are added, and local slots not specified by the class C_{to} that are specified by the class C_{from} are discarded.

The values of local slots specified by both the class C_{to} and the class C_{from} are retained. If such a local slot was unbound, it remains unbound.

The values of slots specified as shared in the class C_{from} and as local in the class C_{to} are retained.

This first step of the update does not affect the values of any shared slots.

Initializing Newly Added Local Slots

The second step of the update initializes the newly added slots and performs any other user-defined actions. This step is implemented by the generic function **update-instance-for-different-class**. The generic function **update-instance-for-different-class** is invoked by **change-class** after the first step of the update has been completed.

The generic function **update-instance-for-different-class** is invoked on two arguments computed by **change-class**. The first argument passed is a copy of the instance being updated and is an instance of the class C_{from}; this copy has dynamic extent within the generic function **change-class**. The second argument is the instance as updated so far by **change-class** and is an instance of the class C_{to}.

The generic function **update-instance-for-different-class** also takes any number of initialization arguments. When it is called by **change-class**, no initialization arguments are provided.

There is a system-supplied primary method for **update-instance-for-different-class** that has two parameter specializers, each of which is the class **standard-object**. First this method

checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared as valid. (See the section "Declaring the Validity of Initialization Arguments" for more information.) Then it calls the generic function **shared-initialize** with the following arguments: the instance, a list of names of the newly added slots, and the initialization arguments it received.

Customizing the Change of Class of an Instance

Methods for **update-instance-for-different-class** may be defined to specify actions to be taken when an instance is updated. If only **:after** methods for **update-instance-for-different-class** are defined, they will be run after the system-supplied primary method for initialization and will not interfere with the default behavior of **update-instance-for-different-class**. Because no initialization arguments are passed to **update-instance-for-different-class** when it is called by **change-class**, the **:initform** forms for slots that are filled by **:before** methods for **update-instance-for-different-class** will not be evaluated by **shared-initialize**.

Methods for **shared-initialize** may be defined to customize class redefinition. See the section "Shared-Initialize" for more information.

Reinitializing an Instance

The generic function **reinitialize-instance** may be used to change the values of slots according to initialization arguments.

The process of reinitialization changes the values of some slots and performs any user-defined actions. It does not modify the structure of an instance to add or delete slots, and it does not use any **:initform** forms to initialize slots.

The generic function **reinitialize-instance** may be called directly. It takes one required argument, the instance. It also takes any number of initialization arguments to be used by methods for **reinitialize-instance** or for **shared-initialize**. The arguments after the required instance must form an initialization argument list.

There is a system-supplied primary method for **reinitialize-instance** whose parameter specializer is the class **standard-object**. First this method checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared as valid. (See the section "Declaring the Validity of Initialization Arguments" for more information.) Then it calls the generic function **shared-initialize** with the following arguments: the instance, **nil**, and the initialization arguments it received.

Customizing Reinitialization

Methods for **reinitialize-instance** may be defined to specify actions to be taken when an instance is updated. If only **:after** methods for **reinitialize-instance** are defined, they will be run after the system-supplied primary method for initialization and therefore will not interfere with the default behavior of **reinitialize-instance**.

Methods for **shared-initialize** may be defined to customize class redefinition. See the section "Shared-Initialize" for more information.